Hierarchical Distributed Reference Counting

Luc Moreau
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
L.Moreau@ecs.soton.ac.uk

Abstract
Massively distributed computing is a challenging problem for garbage collection algorithm designers as it raises the issue of scalability. The high number of hosts involved in a computation can require large tables for reference listing, whereas the lack of information sharing between hosts in a same locality can entail redundant GC traffic. In this paper, we argue that a conceptual hierarchical organisation of massive distributed computations can solve this problem. By conceptual hierarchical organisation, we mean that processors are still able to communicate in a peer to peer manner using their usual communication mechanism, but GC messages will be routed as if processors were organised in hierarchy. We present an extension of a distributed reference counting algorithm that uses such a hierarchical organisation. It allows us to bound table sizes by the number of hosts in a domain, and it allows us to share GC information between hosts in a same locality in order to reduce cross-network GC traffic.

1 Introduction
Massively distributed computing has emerged over the last few years as a potentially powerful paradigm of computation. It has taken different shapes: (i) In the 1-Way experiment [9], multiple sites, communicating over wide-area networks across the American continent, are involved in a distributed computation. (ii) The World Wide Web is regarded by many as a support for planet-wide computing: amongst others, WWW programming [5], object systems on the WWW, such as Java and RMII [36], Globe [14, 18] or W3Objects [16]. (iii) Software agents are autonomous programs, possibly relying on mobility [3, 22] to achieve a task delegated to them; they may cooperate with other agents in order to form agent societies [25]. (iv) Amorphous computing [1] involves a multitude of individuals cooperating together to provide a coherent behaviour.

For a long time, the programming language community has defended the idea that garbage collection is an automatic memory management technique that provides the programmer with a powerful abstraction of memory resources, which cases programming. This idea has been taken on board in many languages, including the recent and widespread language Java. Distributed garbage collection, which extends the idea of automatic memory management to distributed memory, has been the focus of much attention [17, 30].

Distributed reference counting has been a popular implementation technique of distributed garbage collection because it is simple to implement and can be nicely integrated with sequential garbage collectors [2, 24, 28, 30]. Its variant, reference listing [4, 30, 34, 36], associates objects not only with reference counters, but also with the list of hosts that have references to them; reference listing is useful to offer some fault-tolerance.

Designing a garbage collector for massively distributed computations is, however, a challenging task. We have identified two problems that hamper the scalability of distributed garbage collection.

1. Reference listing does not scale. In the presence of massively distributed computations on the Internet, the number of hosts that have access to a given reference can become very high, which requires the garbage collector to maintain very large tables.

2. Locality is not taken into account. Distributed GC algorithms usually offer no mechanism by which sites in a same neighbourhood, e.g., two hosts in a cluster, may share information to reduce GC-related traffic with another distant host, e.g., another cluster on another continent.

The first problem is typical of any algorithm that must maintain tables of hosts; the second one is true for all reference counting algorithms that we have encountered.

Our thesis is that massively distributed computations may be conceptually organised in a hierarchy [19, 38] and that distributed garbage collection may rely on this organisation to overcome the two aforementioned scalability problems. An example of hierarchical organisation derived from network interconnectivity is as follows: at the lowest level, we have a COP, i.e., a cluster of PCs, connected to a laboratory network, itself in contact with a departmental backbone, which takes part in a nation-wide network, from which international connections depart. Other hierarchical organisation is conceivable, more specific to the problem being solved.

For every hierarchy, we identify a distinguished site called the gateway. The intuition of our algorithm is that the gateway acts, in the hierarchy, as the representative of the
rest of the computation; symmetrically, it acts in the rest of the computation, as the representative of the hierarchy. For instance, if a host in a COP has access to a reference, the rest of the computation does not have to identify the precise host, but the COP gateway as holder of the reference. Similarly, a pointer being duplicated between two sites in the UK does not have to be observed by its owner in the US, as long as reference counters are correctly maintained by the UK gateway.

In this paper, we first present a flat and reference-listing variant of our algorithm for distributed reference counting and diffusion tree reorganisation (Section 2). We then convey the intuition of a hierarchical organisation (Section 3). Afterwards, we illustrate the design of hierarchical reference counting by several scenarios (Section 4). This is followed by a formal presentation of the algorithm (Section 5) and some implementation issues (Section 6). The paper ends by a discussion of related work (Section 7) and a conclusion (Section 8).

**Terminology**

This algorithm has been designed as part of NeXeme [25], a distributed implementation of Scheme, based on the message-passing library Nexus [10]. In NeXeme, computations can proceed in distributed memory spaces, called sites. As in Nexus, there is a notion of *global pointer* GP which is a first-class name for an object; a GP specifies a destination to which a communication can be directed via a form of remote method invocation. As far as garbage collection is concerned, GPs are references to possibly remote objects. In addition, NeXeme provides a function *owner* that returns the site that owns the object at which a GP is pointing.

## 2 Flat Distributed Reference Counting

We previously sketched a new algorithm for distributed reference counting [25]. We formalised this algorithm and proved its safety and liveness [24]. In this section, we briefly explain a variant that uses reference listing [4, 30]; we also discuss some of its problems if used in massively distributed computations.

Each site owns two tables, noted *Receive* and *Send* tables, noted _Rec_T_ and _Send_T_ in the algorithm. A Send-table records triples of information: the global pointers that were sent to remote sites, the sites where they were sent to, and the number of times they were sent; according to the terminology of _[4, 30]_, a Send-table maintains a reference listing. Symmetrically, a Receive-table also records triples of information composed of the global pointers that were received from remote sites, the sites that emitted them, and the number of times they were received; in addition, we assume that a GP constructed (and therefore owned) by a site is entered in its Receive-table.

Figure 1.1 shows the situation where a global pointer GP, owned by s1, is copied from s1 to s2, which we model by the message _COPY_ (s1, s2, GP). If the GP is copied for the first time, a new entry is created in the Send-table of s1 for GP; the destination s2 and the initial value 1; for every next copy towards s2, the counter is incremented. Symmetrically, a GP received by s2 from its owner s1 is entered in the Receive-table; multiply receiving a GP from s1 increments the counter.

As the Send-table is implemented as a root of the local garbage collector, the presence of GP in the Send-table prevents its space to be reclaimed on s1. When GP becomes garbage on s2, GP is removed from the Receive-table of s2; then, a decrement message _DEC_ (s2, s1, GP, s2) is sent from s2 to s1, which in turn removes the entry in the Send-table of s1, as displayed in Figure 1.2.

The novelty of our algorithm is exhibited in Figure 1.3, when a site s2, which does not own a GP, sends a copy to a third site s3.

(i) An entry for GP and s3 is added to the Send-table of s2.
(ii) When GP is received by s3 for the first time, an entry is added for GP in its Receive-table. However, the entry records s1, the owner of the GP, and not s2, the GP's emitter.
(iii) A new message _INC-DEC_(s2, s3, GP, s2) is sent from s3 to s1, to inform the owner of the arrival on s3 of a GP originating from s2 (Figure 1.3).
(iv) When receiving the _INC-DEC_ message, the owner s1 adds a new entry for GP and s3 in its Send-table, and then sends a _DEC_(s1, s2, GP, s3) message to s2.
(v) The decrement message sent to s2 decrements the entry in its Send-table, and removes the entry because it becomes null.

The effect of the _INC-DEC_ message (followed by the _DEC_ message) is to reorganise the diffusion tree of GP. GP was diffused from s1 to s2 and then to s3, but the tables are now recording that two copies of GP owned by s1 exist on s2 and s3. The benefit of this reorganisation is that if GP becomes garbage on s2, its space can be reclaimed, whereas in Figure's indirect reference counting [29], a zombie pointer would have to be maintained on s2 as long as GP is used on s1. Note that the algorithm correctness relies on in-order message delivery; indeed, it is essential to prevent a _DEC_ message from overwriting an _INC-DEC_ message, as this may result in an undesirable object reclaiming.

Further copying of GP from s2 to s3, which do not own GP, increase the respective Send and Receive tables (Figure 1.5). Note that it is no longer required to involve the owner with an _INC-DEC_ message, because this only has to be performed the first time the GP is received. At any time, a _DEC_ message may be sent to erase those entries and restore the system in the situation of Figure 1.4.

Thanks to this reorganisation mechanism, this algorithm is able to avoid zombie pointers resulting from computations jumping from node to node. This algorithm however suffers from some defects if used in the context of massively parallel computations [8, 13, 14].

1. **Reference listing does not scale.** The reference listing method requires Send-tables to record all the sites that have access to a given GP. In the presence of the Internet, this may potentially imply very large tables.

2. **The algorithm does not take locality into account.** In Figures 1.3 and 1.4, let us assume that sites s2 and s3 are connected via fast communication lines, and that communication with s2 is slow. The technique to reorganise diffusion trees may force a communication with a site with slow communication. Similarly, in Figure 1.5, sites s2 and s3 may be poorly connected and _DEC_ messages from s3 could be grouped with other messages from sites in the neighbourhood of s2. The lack of locality awareness also hampers the scalability of other algorithms, such as [4, 29, 30].

In the next section, we present a hierarchical organisation of sites that allows us to avoid these two problems.
Figure 1: Flat Distributed Reference Counting
3 Hierarchical Organisation

Sites that are in the same proximity or in a same logical organisation are said to be grouped in *domains*; for instance, COPs, i.e. clusters of PCs, or NOWs, i.e. networks of workstations, may be regarded as domains. As domains are designed to follow a hierarchical organisation, domains are defined as sets of sites or subdomains. Sites (or subdomains) that belong to a domain are said to be *siblings*. We associate each domain with a distinguished site that acts as a *gateway* to the parent domain. For instance, in Figure 2, s₁, s₂, s₃ are siblings and belong to the same domain A, with gateway s₄; sites s₅, s₆, s₇ are siblings and belong to domain B whose gateway is s₉.

For *garbage collection* purposes, we regard the gateway as a key component of a domain: (i) The gateway s₄ of domain A acts as the representative in domain B of all the sites of domain A. In other words, elements of domain B are only aware of the existence of s₄ and not of its children sites s₁, s₂, s₃. (ii) Symmetrically, the gateway s₅ of domain A acts as the representative in domain B of all the sites that do not belong to domain A. In other words, sites of A are only aware of their sibling sites and of s₄ which acts as a proxy for all the other sites.

We want to use this hierarchical organisation in order to guarantee that: (i) reorganisation as in Figure 1.3 can be hidden by a domain gateway; (ii) table sizes for a given GP are bounded by domain sizes. Figure 2 presents a conceptual way of implementing such a hierarchical organisation: gateways between domains could also act as message forwarders. For instance, in order to send a message from s₁ to s₈, s₁ sends it to the gateway s₄ because s₆ is outside domain A, which in turn forwards it to s₉ for the same reason, etc.

In practice such a solution is *not acceptable* because it potentially delays the sending of messages as it involves several domain gateways in message forwarding¹. However, from a theoretical point of view, this solution is suitable because a GP copied from s₁ to s₄ would create an entry for s₄ in the Send-table of s₁, and reciprocally for the Receive-table of s₄. Such a property would also hold for every gateway, which would guarantee that table sizes are bounded by domain sizes.

In the next Section, we present an extension of the flat GC that does not increase the cost of sending mandatory messages, but maintains a hierarchical organisation as we just described.

4 Hierarchical Distributed Reference Counting

As in every garbage collector design, it is essential to minimize the impact of garbage collection activity on mandatory computation. In particular, remote method invocation, which copies global pointers, should be executed as efficiently as possible. Therefore, instead of hierarchical message forwarding of Figure 2, we prefer the organisation of Figure 3, where we see that the mandatory message is directly sent from s₁ to s₈, but a conceptual hierarchy is kept for GC purposes. Two new messages, introduced to inform gateways of cross-domain messages, are asynchronously sent to gateways of the hierarchy; they are defined in terms of three sites s₁, s₂, s₅:

- With $DOMSEND(s₁,s₁,GP,s₅)$, site (or subdomain) s₁ informs its gateway s₅ that GP was sent to s₅ belonging to another domain.

- With $DOMRECV(s₁,s₅,GP,s₅)$, site (or subdomain) s₁ informs its gateway s₅ that GP was received from s₅ belonging to another domain.

In Figure 3, $DOMSEND$ and $DOMRECV$ messages are repeatedly sent till they respectively reach s₉ and s₆, such that s₁ is an ancestor of s₅, s₃ is an ancestor of s₆, and s₅ is sibling of s₆.

When messages are exchanged within a domain, the flat garbage collection algorithm of Figure 1 is used. If global pointers are sent to or received from sites belonging to other domains, then Figure 4 summarises the hierarchical protocol.

In Figure 4.1, site s₂ sends a copy of GP to s₃, a site outside the domain A. A new entry is added to the Send-list of s₂, with GP and the gateway s₉, as if the message had been routed via the gateway s₉. In addition, a message $DOMSEND$ is sent to the gateway, which acts as if it was forwarding the message. Gateways also maintain Receive

¹We assume here that we are using a network layer, such as TCP/IP, which already efficiently performs routing between networks. Every GC gateway acts as a forwarding protocol. Note also that hierarchical domains may also be different from the network organisation.
and Send tables: (i) Within domain $A$, the flat GC algorithm applies. If this is the first time that $s_A$ receives a copy of $GP$, and if $s_1$ owns $GP$, then $INC, DEC$ and $DEC$ messages are sent in the usual triangular mode, and the Receive table of $s_A$ records that $GP$ comes from $s_1$. (ii) If $s_3$ (or one of its gateways) is a sibling of $s_A$, then the Send-table entry of $s_A$ records this gateway. Otherwise, if $s_3$ (or one of its gateways) is not a sibling of $s_A$, then the same mechanism applies recursively, and $s_A$ sends a $DOM, SEND$ message to its gateway, etc. Figure 3 illustrates a succession of such $DOM, SEND$ messages. A $DOM, SEND$ message is successively sent to gateways of the hierarchy till it reaches a gateway $s_0$ (in Figure 3), which is a sibling of a gateway $s_0$ itself and an ancestor of the $GP$ receiver $s_a$.

Figure 4.2 describes the symmetric situation where a site $s_1$ receives a copy of a $GP$ from $s_3$ outside the domain $A$. The system behaves as if $GP$ had been received from the gateway $s_A$. Hence, a $DOM, REC$ is sent to the gateway $s_4$, which implies that the Send-table of $s_A$ records that $GP$ is sent to $s_4$ and symmetrically for the Receive-table of $s_1$. If it is the first time that $GP$ is received by $s_1$ and if $GP$ is owned by $s_2$, a sequence of $INC, DEC$ and $DEC$ messages is required as in the flat GC algorithm2. If $s_3$ (or one of its gateways) is not a sibling of $s_A$, then we proceed recursively as illustrated in Figure 3. A $DOM, REC$ message is successively sent to gateways of the hierarchy till it reaches a gateway $s_0$ (in Figure 3), which is a sibling of a gateway $s_0$, itself and an ancestor of the $GP$ emitter $s_A$.

In summary, gateways act as the Receive- and Send-tables of domains. The $DOM, SEND$ or $DOM, REC$ messages consist of incrementing the Send or Receive tables in the gateway, respectively, possibly entailing a triangular reorganisation as in the flat version of the algorithm.

5 The Algorithm

We formalise the algorithm using an abstract machine, called the HGC machine, whose configuration is defined at

\[ DOM, SEND(s_1, s_2, GP, s_3) \]

\[ DOM, REC(s_1, s_2, GP, s_3) \]

\[ COPY(s_1, s_2, GP) \]

\[ \text{Figure 3: Hierarchical Update of Tables} \]

The top of Figure 5. A configuration $H$ is a tuple composed of a set of sites, a set of global pointers, Receive and Send tables for every site, and a bag of messages. Receive and Send tables are functions which for a site $s_1$, a $GP$, a site $s_2$ respectively return the number of times $GP$ was received by $s_1$ from $s_2$, or sent from $s_1$ to $s_2$. The bag of messages denotes messages that are in transit in the system, i.e. already posted but not yet handled.

We write $owner(GP)$ to denote the site that owns $GP$. Initially, all table entries are zero, except for hosts that own $GP$: $sendLT(s_1, GP, s_2) = 0, \forall s_1, s_2$.

$recLT(s_1, GP, s_2) = 1$ if $owner(GP) = s_1$.

$\forall s_1, s_2, GP, \text{otherwise}.$

We allow HGC-configurations to perform four transitions. Let $H_1$ be $(S, G, sendLT, recLT, M)$, and let us assume that $s_1, s_2, s_3 \in S, GP \in G, m \in M$, then:

\[ H_1 \Rightarrow MAKECOPY(s_1, s_2, GP) \Rightarrow H_2 \]  

\[ (make-copy) \]

\[ H_1 \Rightarrow RECEIVE(m) \Rightarrow H_2 \]  

\[ (receive) \]

\[ \text{if } m = DEC(s_1, s_2, GP), \text{then} \]

\[ INC, DEC(s_1, s_2, GP, s_3) \notin M, \forall s_3 \]

\[ H_3 \Rightarrow RELEASE(s_1, GP) \Rightarrow H_2 \]  

\[ (release) \]

\[ H_1 \Rightarrow DELETE(s_1, s_2, GP) \Rightarrow H_2 \]  

\[ (delete) \]

The configuration transformers $MAKECOPY$, $RECEIVE$, $RELEASE$ and $DELETE$ are defined in Figures 5 and 6. In Figure 5, if $H_1 = (S_1, G_1, sendLT, recLT, M_1)$, $post(m)$ denotes the configuration $H_2$ such that $H_2 = (S_2, G_2, sendLT, recLT, M_2 \cup \{m\})$. Similar notational conventions are used for assignments to Send and Receive tables. In the rules below, we note the side-condition of ($receive$), which may be implemented by in-order message delivery. We also assume that transitions are executed atomically.

In addition, in order to model the hierarchical domain organisation, we use the following relations:

- $\text{parent}(x, y)$: node $y$ is parent of node $x$.
- $\text{descendant}(x, y)$: node $y$ is a descendant of node $x$, if $x = y$ or $\exists z, \text{parent}(z, x) \land \text{descendant}(z, y)$. 

5
- **sibling(x,y)**: node y is a sibling of node x, if \( \exists z, \) parent(x,z) = parent(y,z).

- **proxy(x, y)**: the representative of y in the domain of x. For the time being, we consider a static hierarchical organisational: the proxy relation is defined as follows:

\[
\text{proxy}(x, y) = \begin{cases} 
  y & \text{if sibling}(x, y) \\
  z & \text{if } \neg \text{sibling}(x, y) \land \text{descendant}(x, y) \\
  \text{parent}(z, x) & \text{if } \neg \text{descendant}(x, y) \land \text{parent}(z, x)
\end{cases}
\]

Let us first consider the particular case where all sites belong to the same domain, which means that sibling(x,y) holds and that proxy(x,y) = y for any x,y. In such a situation, Figure 5 describes the algorithm for "flat" garbage collection.

Rule (make-copy) associated with configuration transformer MAE_COPIY models the actions that the site s1 has to take before sending a copy of GP to s2. The sendLT is updated and a COPY message is posted.

Rule (receive) and its associated configuration transformer RECEIVE describe how messages are received. When s2 receives a COPY message from s1, a triangular reorganisation is initiated if it is the first time GP is received, i.e. the entry in the receive table for the owner s2 is empty. The triangular reorganisation involves sending an INC-DEC message to the owner, which is followed by a DEC message.

An entry in the receive table pointing at a site that is not the owner of a GP may always be cleared by rule (release), which sends a decrement message to the site. Finally, a local garbage collector that proves that a GP has become garbage on site s initiates the transition (delete); such a transition can only be fixed if Receive and Send tables (except for the owner s2) are empty, and it results in a DEC message sent to the owner.

If the sites involved in the transitions do not belong to the same domain, rules of Figure 5 remain still applicable, but are now involving proxies of the sites. For instance, receiving a COPY message from s1, which does not belong to the same domain as s2, potentially results in a triangular reorganisation with s2 and s_p, respectively proxies of s1 and the owner (cf. Figure 4.2). In addition, a DOM_RECV message is sent, when s2 is not sibling with (the proxy of) s1.

Figure 6 displays how the two messages for hierarchical GC are handled. Receiving a message DOM_RECV(s1,s2,GP,s_p) is similar to receiving a message COPY(s1,s2,GP). The only difference is that the gateway s2 has to act as if it was forwarding the message originating from s_p to s1 (as in Figure 2). As a result, the entry for s1 in the sendLT of s2 has to be incremented.
\[
\begin{align*}
S & = \{s_0, s_1, \ldots, s_n\} & \text{(Site)} \\
GP & \in G = \{GP_0, GP_1, \ldots\} & \text{(Global Pointer)} \\
m & \in \text{Msg} := \text{COPY}(s_1, s_2, GP) \upharpoonright \text{DEC}(s_1, s_2, GP, s_0) \upharpoonright \text{INC}_n \text{DEC}(s_1, s_2, GP, s_0) \upharpoonright \text{DOM}_n \text{SEND}(s_1, s_2, GP, s_0) \upharpoonright \text{DOM}_n \text{RECV}(s_1, s_2, GP, s_0) & \text{(Message)} \\
M & : \text{BagOf}(\text{Msg}) & \text{(Pool of Messages)} \\
\text{sendLT} & : S \times G \times S \rightarrow \text{IN} & \text{(Send Tables)} \\
\text{rec\text{-}T} & : S \times G \times S \rightarrow \text{IN} & \text{(Receive Tables)} \\
\mathcal{H} \in \text{Config} & := \langle G, G \leftrightarrow S, \text{send\text{-}LT}, \text{rec\text{-}T}, M \rangle & \text{(HGC-Configuration)}
\end{align*}
\]

\[\text{MAKE\text{-}COPY}(s_1, s_2, GP) \quad \text{if} \quad s_1 \neq s_2 \land \text{rec\text{-}T}(s_1, GP, \text{proxy}(s_1, \text{owner}(GP))) > 0 \]
\[
\begin{align*}
& \text{let} \quad s_p = \text{proxy}(s_1, s_2) & & \text{/} s_1 \text{ prepares to copy } GP \text{ to } s_2 \\
& \text{sendLT}(s_1, GP, s_p) := \text{sendLT}(s_1, GP, s_p) + 1; \\
& \text{post}(\text{COPY}(s_1, s_2, GP)); \\
& \text{if} \quad \neg\text{sibling}(s_1, s_2) \quad \text{then} \\
& \quad \text{post}(\text{DOM}_n \text{SEND}(s_1, s_p, GP, s_2)) \\& \end{align*}
\]

\[\text{RECEIVE}(\text{COPY}(s_1, s_2, GP)) \quad \text{if} \quad s_1 \neq s_2 \]
\[
\begin{align*}
& \text{let} \quad s_p = \text{proxy}(s_2, s_1) & & \text{/} s_2 \text{ receives a copy of } GP \text{ from } s_1 \\
& \quad s_g = \text{proxy}(s_2, \text{owner}(GP)) \\
& \text{if} \quad \text{rec\text{-}T}(s_2, GP, s_g) = 0 \quad \text{then} \\
& \quad \{ \text{rec\text{-}T}(s_2, GP, s_g) := 1; \\
& \quad \text{post}(\text{INC}_n \text{DEC}(s_2, s_3, GP, s_1)) \quad \text{if} \quad s_p \neq s_g \land s_2 \neq s_g \} \\
& \text{else} \\
& \quad \{ \text{rec\text{-}T}(s_2, GP, s_g) := \text{rec\text{-}T}(s_2, GP, s_g) + 1 \}; \\
& \text{if} \quad \neg\text{sibling}(s_2, s_3) \quad \text{then} \\
& \quad \{ \text{post}(\text{DOM}_n \text{RECV}(s_2, s_p, GP, s_1)) \} \\& \end{align*}
\]

\[\text{RECEIVE}(\text{INC}_n \text{DEC}(s_1, s_2, GP, s_3)) \]
\[
\begin{align*}
& \text{let} \quad s_p = \text{proxy}(s_2, s_3) & & \text{/} s_2 \text{ receives an INC\text{-}DEC message} \\
& \text{sendLT}(s_2, GP, s_1) := \text{sendLT}(s_2, GP, s_1) + 1; & & \text{/} s_2 \text{ which received } GP \text{ from } s_3 \\
& \text{post}(\text{DEC}(s_2, s_p, GP, s_1)); \\
& \end{align*}
\]

\[\text{RECEIVE}(\text{DEC}(s_1, s_2, GP, s_3, n)) \]
\[
\begin{align*}
& \{ \text{sendLT}(s_2, GP, s_3) := \text{sendLT}(s_2, GP, s_3) - n \} & & \text{/} s_2 \text{ receives an DEC message from } s_1 \\
& \end{align*}
\]

\[\text{RELEASE}(s_1, s_2, GP) \quad \text{if} \quad s_1 \neq s_2 \land s_2 \neq \text{proxy}(s_1, \text{owner}(GP)) \\
\]
\[
\begin{align*}
& \text{rec\text{-}T}(s_1, GP, s_2) > 0 \\
& \text{rec\text{-}T}(s_2, GP, \text{proxy}(s_1, \text{owner}(GP))) > 0 \\
& \{ \text{post}(\text{DEC}(s_1, s_2, GP, s_1)); \text{rec\text{-}T}(s_1, GP, s_2) := 0 \} & & \text{/} s_1 \text{ annihilates receive table entry} \\
& \end{align*}
\]

\[\text{DELETE}(s_2, GP) \quad \text{if} \quad s_g = \text{proxy}(s, \text{owner}(GP)), \\
\quad s \neq s_2, \\
\quad \forall s_2 \quad \text{sendLT}(s_2, GP, s_2) = 0, \\
\quad \forall s_2 \neq s_1 \neq s_g \quad \text{rec\text{-}T}(s_2, GP, s_2) = 0 \\
\]
\[
\begin{align*}
& \{ \text{post}(\text{DEC}(s, s_g, GP, s)); \text{rec\text{-}T}(s, GP, s_g) := 0 \} & & \text{/} GP \text{ becomes garbage on } s \\
& \end{align*}
\]

---

Figure 5: Flat Garbage Collection
RECEIVE(DOM_RECV(s1, s2, GP, s3))  
\{  
  \text{let } s_0 = \text{proxy}(s_2, s_3)  
  s_0 = \text{proxy}(s_2, \text{owner}(GP))  
  \text{if } \text{rec}(s_2, GP, s_0) = 0 \land \neg \text{descendant}(s_0, s_2) \text{ then}  
  \{    
    \text{rec}(s_2, GP, s_0) = 1;  
    \text{post}(INC\_DEC(s_2, s_3, GP, s_1)) \text{ if } s_3 \neq s_2 \land s_2 \neq s_0  
  \}  
  \text{else}  
  \{    
    \text{rec}(s_2, GP, s_0) = \text{rec}(s_2, GP, s_0) + 1;  
    \text{send}(s_2, GP, s_1) = \text{send}(s_2, GP, s_1) + 1;  
    \text{if } \neg \text{sibling}(s_2, s_0) \text{ then}  
    \{ \text{post}(DOM\_RECV(s_2, s_3, GP, s_3)) \}  
  \}  
\}  

RECEIVE(DOM\_SEND(s_1, s_2, GP, s_3))  
\{  
  \text{let } s_0 = \text{proxy}(s_2, s_3)  
  \text{if } \text{rec}(s_2, GP, s_0) = 0 \land \text{descendant}(s_0, s_2) \text{ then}  
  \{    
    \text{rec}(s_2, GP, s_0) = 1;  
    \text{post}(INC\_DEC(s_2, s_3, GP, s_1)) \text{ if } s_3 \neq s_1  
  \}  
  \text{else}  
  \{    
    \text{rec}(s_2, GP, s_0) = \text{rec}(s_2, GP, s_0) + 1;  
    \text{send}(s_2, GP, s_1) = \text{send}(s_2, GP, s_0) + 1;  
    \text{if } \neg \text{sibling}(s_2, s_0) \text{ then}  
    \{ \text{post}(DOM\_SEND(s_2, s_3, GP, s_3)) \}  
  \}  
\}  

Conceptually, when a gateway $s_2$ receives a message $DOM\_SEND(s_1, s_2, GP, s_3)$ from $s_1$, it must act as if it was forwarding the message to its destination $s_3$. Therefore, the receive table records the arrival of $GP$ from $s_1$ (or $s_0$ via the usual triangular reorganisation). In addition, the entry for the proxy of $s_3$ is incremented in the Send-Table of $s_2$.

Our hierarchical organisation solves the problem of the flat GC algorithm:

- **Locality:** Gateways are hiding reorganisations within their domains. For instance, the copy of $GP$ owned by a site outside the domain results in a triangular exchange as described by Figure 7, where the gateway acts as a representative for the $GP$'s owner. In addition, as the gateway $s_A$ of Figure 7 "centralises" GC information about $GP$ for domain $A$, we are able to send a decrement message to its owner only when $GP$ has become garbage on all sites of domain $A$. Such sharing of information in a domain allows us to reduce GC traffic across domains.

- **Table Size:** per global pointer, a table now has a maximum number of entries given by the size of the domain. Indeed, every access to a Send-Table (and similarly for a Receive-Table) in the algorithm of Figures 5 and 6 is of the form $\text{send}(s_1, GP, s_0)$, where $s_0$ was defined as $\text{proxy}(s_1, s_3)$, for any site $s_0$. So, entries in Send-Tables always refer to sites in the same domain.

---

**Figure 6: Hierarchical Garbage Collection**


**Figure 7: Inside Domain Triangular Reorganisation**

6 Implementation

This distributed GC algorithm has been designed as part of NeXen [25] a distributed extension of Scheme based on the message-passing library Nexus [10]. The flat algorithm has been fully implemented and tested, but (at the time of writing), the hierarchical implementation is still in progress.

An important aspect of the implementation is the design of an efficient function sibling. Currently, we define a hier-
archical pointer as a structure composed of a global pointer and an access path, which is a data-structure representing the path of its owner in the hierarchy. By default, the access path is set to nil, which means that the hierarchical pointer has not exited its domain. Every time a hierarchical pointer is serialised, we have to compare its access path with the access path of the message destination. If the access paths are equal, then the hierarchical pointer is being sent to its domain. Otherwise, from the common path prefix, one can derive the gateways to which the DOM_SEND and DOM_RECV messages must be sent.

For the time being, gateways are built as distinguished processes. We are investigating how to implement them as regular nodes, which could also take part in the computation. So far, we have considered a static hierarchy. We are also studying ways of specifying the hierarchy dynamically and of changing gateways at runtime according to the load of the system.

We have presented here an abstract algorithm, which can be optimised in several ways. (i) Several DOM_SEND (or DOM_RECV) could be merged together, in the same spirit as DEC messages that contain a counter value. (ii) One message could be avoided in Figure 4.2 by recognising this particular situation, to the detriment of algorithm readability; for instance, the DOM_RECV and DEC messages could be combined in a new message that would follow the INC-DEC message. (iii) We can design different strategies for sending GC messages. For the time being, a regular site sends GC messages as soon as it becomes idle. On the other hand, gateways infrequently send messages to their siblings in order to avoid cross-domain traffic.

7 Related Work

Reference-counting garbage collection was initially developed for uniprocessor systems [6]. Its principle is as follows: every time a pointer is copied or deleted, a reference count is respectively incremented or decremented. It might seem that this algorithm can be extended straightforwardly to distribution by using two control messages INC and DEC that act on the reference counter residing on the owner of the pointer. Unfortunately, non-causal message delivery may reset the counter even though remote references may still be active. Correct solutions to this problem have been proposed, including weighted reference counting [2, 8, 30] and its optimised version [7], generational reference counting [12], indirect reference counting [28, 29].

Other tripartite exchange of messages including the site that emitted a GP, the receiver of the GP, and its owner can be found in the literature, in particular by Lermen and Mauter [21, 35], and by Birrel et al. [4]. Our algorithm differs from theirs by the direction in which messages are exchanged and because our solution only requires a tripartite exchange the first time a GP is received. Intuitively, our solution preserves causality because the owner sends a DEC message only after having processed an INC-DEC message which has increased a reference counter.

However, our algorithm has another major benefit as it is able to recognize diffusion trees: when GC messages are all processed, the diffusion tree is completely flattened, and every site owning a GP directly “depends" from its owner. In the presence of mobile computations jumping from site to site, this allows sites to reclaim the space that was occupied by a mobile program, hereby avoiding zombie references as in indirect reference counting [28]. To the best of our knowledge, Shapiro, Diekman, and Plainfosse [34, 35] were the first to address the issue of short-circuiting chains of pointers. They regard migration as a primitive notion to be supported by the GC; in this paper, we do not deal with migration, however, we have showed that support for mobility could be added as an extra layer, like a library, on top of the current garbage collection algorithm [24].

The distributed collector of Java with Remote Method Invocation [36] is derived from Birrel’s network objects [4]. In addition, Java uses a mechanism of leases, by which sites having pointer copies are forced to regularly renew their lease. Such a mechanism supports fault-tolerance and could also be implemented with our algorithm.

Let us note that none of the previously mentioned algorithms is based on a hierarchical organisation as presented in this paper. Therefore, those using reference listing [4, 35, 36] potentially have to manage very large tables in Internet-wide computation. Furthermore, they are not able to share garbage collection information within a neighbourhood, which prevents them from optimising GC information transfer on a local basis, such as per cluster or per network.

We are accustomed to hierarchical memories in uniprocessor systems and memory management has been studied in this particular case [40]. We can regard our schema as a hierarchical organisation of a distributed memory. Other hierarchical organisations have been brought forward. Lang, Queinnec, and Piquer “Garbage collecting the World" uses a hierarchy of nodes that are willing to cooperate for garbage collecting; such a similar approach is also adopted in [38] in order to provide scalability. Queinnec [31] also suggests to cluster sites so that they can present a single clock to the rest of the world; clocks are used to provide a causally-coherent distributed memory.

Rodrigues and Jones [32] dynamically identify groups of processes that will collaborate to reclaim distributed cyclic garbage. Their groups provide locality as communication related to the garbage collection activity is only necessary between members of the group. The same authors [33] also explain how groups that have independently initiated a collection on the same cycle may merge together. Mathewaru and Leskov [23] use back tracing [11] to determine if an object is garbage. Back tracing as opposed to forward tracing preserve locality of the tracing process.

Reference counting garbage collection is only able to reclaim acyclic data structures. Other algorithms may be combined with ours in order to collect cycles, such as Le Fessant, Fumaro, and Shapiro’s [20], Rodrigues and Jones’ [32], or Lang, Queinnec and Piquer’s [19]. The latter seems to be particularly appropriate because it also relies upon a hierarchical organisation of sites that cooperate to eliminate cycles between themselves. Gateways in our algorithm contain Send and Receive tables for a domain and can be used to perform a collaborative garbage collection of the domain. The distributed variant of the Tram GC [15] is also able to collect cycles; it combines a reference-counting style pointer-tracking mechanism with a substitution protocol.

8 Conclusion

We have presented a hierarchical organisation for distributed reference counting. Such an approach is particularly suitable for Internet-wide programming because it is able to abstract a whole domain of sites by a single host, which acts as its
representative for garbage collection purposes. Such a hierarchical schema allows us to give bounds on the size of tables involved in reference listing and to reduce cross domain GC traffic.

This algorithm is being implemented as part of NeXem, a distributed implementation of Scheme [25], offering some support for mobile computations [24] and distributed resource control [27]. We foresee two other applications for this algorithm. First, in [25], we associate reference counters with WWW documents, in order to build an agent architecture that offers link integrity in a publishing environment; the new reference counting algorithm may be used to improve scalability of the system. Second, in [27], we present a distributed model of resource control, suitable for agent-style applications. We believe that the present algorithm can be applied to resource control in order to facilitate resource management among sites in a same locality.

9 Acknowledgement

This research was supported in part by the Engineering and Physical Sciences Research Council, grant GR/K30773. We are grateful to Stuart Maclean, Danilo Michaelides, Christian Queinnec, and the anonymous referees for their comments.

References


[34] Marc Shapiro, Peter Dickman, and David Plainfoss, Robust, Distributed References and Acyclic Garbage Collection. In Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1992.


